Set-Oriented Constructs for Rule-Based Systems

Douglas N. Gordin
AT&T Bell Laboratories
184 Liberty Corner Road
Warren, NJ 07050
gordin@melmac.att.com

Alexander J. Pasik
Department of Computer Science
Columbia University
New York, NY 10027
pasik@cs.columbia.edu

Abstract

Set-oriented constructs for forward chaining rule-based systems are presented in this paper. These constructs allow arbitrary amounts of data to be matched and changed within the execution of a single rule. Second order tests on the data can be included in the match. The ability of a single rule to directly access all of the data to be manipulated eliminates the need for unwieldy control mechanisms and marking schemes. Adding this expressivity to rule-based languages enhances their value to expert system developers and their capabilities as database programming languages. Additionally, these set-oriented constructs can provide a basis for more efficient implementations of rule-based systems, for both the traditional memory-based systems and the emerging disk-based ones. The work described has been implemented using an extended version of the Rete algorithm.

AI Topic: Knowledge Representation, Rule-based systems.
Language/Tool: OPS5.
Effort: Proprietary Information.
Impact: Set-oriented constructs in rule-based languages will enable fast prototyping of expert systems and provide new directions for integrating rule-based systems and database systems.

1. Introduction

Set-oriented constructs for forward chaining rule-based systems allow arbitrary amounts of data to be matched and changed within the execution of a single rule. Second order tests on the data can be included in the match. These enhancements provide a succinct method to perform commonly required operations while adding new opportunities for efficiency optimizations.

The ability of a single rule to access all of the relevant data eliminates the need for unwieldy control mechanisms and marking schemes. Similarly, the direct specification of second order tests eliminates the need to compute these values. Such abilities are standard within database systems, but were formerly not integrated with rule-based programming. Adding this expressivity to rule-based languages enhances their value to expert system developers and their capabilities as database programming languages.

It is hoped that these set-oriented constructs will provide a basis for more efficient implementations of rule-based systems, both for the traditional memory-based systems and the emerging disk-based ones. The direct expression of set operations allows compilers to optimize these operations. For example, a parallel architecture could perform an operation on the members of a set in parallel. Furthermore, research has shown that a limiting factor for parallelization of the Rete network is the number of operations done per rule firing [Gupta 1984, Miranker 1986, Pasik 1989]. The number of actions in a set-oriented rule should be substantially greater, providing the ability to increase parallelism.

All of the work described has been implemented within C5 [Vesonder 1988], a language that is an upwardly compatible superset of OPS5 [Forgy 1981]. The introduction of the set-oriented changes were made in a way that does not degrade the performance when executing regular OPS5 programs, and has been implemented using an extended version of the Rete algorithm [Forgy 1982].

2. Related Literature

The fields of rule-based systems, databases, and programming languages have been converging. This work relates both to rule-based programming languages and adding rules to databases.

2.1. OPS LANGUAGES

In research related to OPS rule languages, matching sets of working memory elements (WMEs) on the left-hand-side (LHS) has been proposed before to increase the amount of parallelism in production systems [van Biema et al. 1986]. However, this proposal was never developed in detail.

Iterating over sets on the right-hand-side (RHS) is proposed as an addition to OPS5 in the YES/OPS work at IBM [Schor et al. 1986]. That work differs in that the matching is performed on the RHS rather than the LHS, hence they cannot do any second order aggregate matching.

2.2. RULE-BASED SYSTEMS AS DATABASE MANAGEMENT SYSTEMS

Some investigators are attempting to build rule-based systems that are disk based rather than memory based [Sellis et al. 1988]. This approach attempts to provide facilities for handling much larger amounts of data by matching on disk. However, this approach is undermined by

1This paper assumes knowledge of OPS5 as described in [Forgy 1981].
the tuple-orientation of most rule-based systems. Greater optimization could be performed for set-oriented rules since they can cause large aggregate changes to the database.

Another proposal has been to use SQL queries for the LHS and having the RHS actions apply uniformly to the entire relation matched [Delcambre and Etheredge 1988]. This proposal, RPL, is similar to ours in that the LHS matches an entire relation rather than just one tuple of the relation. Moreover, their LHS expressivity is greater since SQL is relationally complete. However, our set-oriented constructs provide a natural extension of OPS5 that maintains programming style and can be implemented in an efficient manner. In addition, the RHS constructs presented below are more flexible that that of RPL. Other work extends SQL databases with rule-based functions [Widom and Finkelstein 1990, Stonebraker et al. 1990].

3. OPS5 as a Database Language

A number of researchers have chosen to describe OPS5 as a database programming language [Miranker 1986, Delcambre and Etheredge 1988, Tzvieli 1988]. This analogy is established by noticing:
- Working memory (WM) is a relational database with one important difference: Each WME has a time tag that uniquely identifies it.
- The LHS of productions are relational queries that perform selections and joins. Each comparison to a constant performs a selection. The appearances of a pattern variable (PV) in multiple condition elements (CEs) causes a join. Thus, each LHS generates a relation, possibly empty, of joined WMEs that satisfy the conditions.
- Each tuple in the relation is an instantiation.
- RHS actions update the database.

The concepts of rules, WM, and the conflict set of instantiations are illustrated in Figure 1. WM contains team members on two different teams. The rule compete generates all possible competitions between members of the two teams. Note that in a relational database system, the six instantiations of the compete rule would be contained in a single relation.

4. Adding Set-Oriented Constructs to the Left-Hand-Side

Set-oriented constructs add the ability to specify set-oriented variables and the ability to specify aggregate tests to be performed on those variables.

4.1. SET-ORIENTED CONDITION ELEMENTS AND PATTERN VARIABLES

A set-oriented CE (designated by square brackets) directs the interpreter to include all of its consistent matches within the instantiation. In contrast, regular instantiations only contain a single matched WME for each CE. Hence, a LHS that only contains set-oriented CEs will only produce one instantiation (see Figure 2, compete1). This instantiation will contain the entire relation generated by the LHS. However, a LHS can contain both set-oriented and regular CEs. In this case, the regular CEs can be thought of as partitioning the relation induced by the LHS into smaller relations. Alternatively, the set-oriented CEs can be seen as combining the regular instantiations into aggregated instantiations (see Figure 2, compete2).

![Figure 1: OPS5 Rule, Working Memory, and Conflict Set](image)

A PV is set-oriented if it occurs within a set-oriented CE. These PVs do not have a scalar binding as do regular PVs, rather their domains are specified by the set of values occurring in the WMEs satisfying their CEs (see Figure 2, compete1). When a set-oriented PV occurs in two set-oriented CEs, the domain is reduced to the consistent values of the domains (in database terms a join is performed).

When a PV occurs in both a set-oriented CE and a regular CE it is bound to a scalar value, namely the value occurring in the WME matching the regular CE. Additionally, it may be necessary to specify that a PV only occurring in set-oriented CEs be non-set-oriented. The scalar clause lists PVs that should be non-set-oriented even though they only occur in set-oriented CEs. The effect of this is to partition by value the relation induced by the LHS into separate instantiations.

4.2. AGGREGATE OPERATORS ON THE LEFT HAND SIDE

The addition of aggregate operators on the LHS eases programming by adding expressive power and allows for efficiency optimizations. If OPS5 programs need to act based on the cardinality of a set or an average of a value, it needs to cycle through all the members of that set calculating the second order value. With aggregate operators, this value can be directly accessed. Efficiency
optimizations can be performed since the aggregate operation has been explicitly specified.

```
(p compete1
(player ^name <n1> "team A"
[player ^name <n2> "team B"
-->
1 Instantiation:
1: player A Jack 3: player B Sue
1: player A Jack 4: player B Jack
1: player A Jack 5: player B Sue
2: player A Janice 3: player B Sue
2: player A Janice 4: player B Jack
2: player A Janice 5: player B Sue

(p compete2
[player ^name <n1> "team A"
[player ^name <n2> "team B"
-->
3 Instantiations:
1: player A Jack 3: player B Sue
1: player A Jack 4: player B Jack
1: player A Jack 5: player B Sue
2: player A Janice 3: player B Sue
2: player A Janice 4: player B Jack
2: player A Janice 5: player B Sue

Figure 2: Set-Oriented LHSs and Instantiations
```

The aggregate operators on the LHS include the standard ones from SQL namely, count, min, max, sum, and avg. Currently, only the count operator has been implemented.

5. Adding Set-Oriented Constructs to the Right-Hand-Side

The division of a rule into LHS and RHS breaks up the specification of a relation from the actions to be performed on it. Formerly, this division was obscured by the inability of a rule to access the entire relation that its LHS defined. The addition of set-oriented constructs allows that relation or any part of it to be accessed on the RHS.

The interpreter's control strategy is also affected by the set-oriented constructs. Here, if any part of the instantiation changes, the instantiation is again eligible to fire (cf: [Widom and Finkelstein 1990]).

There are two types of capabilities that have been added to the RHS to access set-oriented PVs or CE. The first is aggregate operations on an entire set such as set-remove and set-modify. The second is an iterator that executes its body on each subset of the instantiation, having a distinct value for a specified set-oriented variable.

The foreach iterator has a designated set-oriented variable and a block of statements. Foreach breaks up the instantiation's relation into subrelations (or subinstantiations) where each subinstantiation has only one value for the iterator variable. This is similar to performing an SQL group-by on that variable. The foreach iterator accesses the items in ascending, descending, or default order. By default, the values are considered in the order in which they would have occurred as separate instantiations in the conflict set.

The iterator's statements is executed for each subinstantiation. This process is described further by discussing separately the two cases of the iterator variable, namely, when it is a set-oriented PV or CE.

5.1. THE FOREACH OPERATOR ON SET-ORIENTED PATTERN VARIABLES

Using a set-oriented PV as an iterator partitions the instantiation by value. The iterator executes its block of statements once for each unique value in the PV's domain. During this execution, the instantiation is reduced to tuples where the PV's attribute is equal to the current value. The iterator variable can now be accessed as a regular PV bound to the current value. Note that the relations formed during the iteration also could have been created if the iterator variable had been specified as scalar in the LHS. However, the subinstantiations would have been different instantiations, rather than present during the execution of iterator. Hence, by matching on a set of values and iterating over them, subinstantiations are made accessible in a single rule firing that would otherwise have been formed into separate instantiations requiring multiple rule firings.

```
(p GroupByTeam
[player ^team <t> ^name <n>]
-->
(foreach <t>
(write <t>)
(foreach <n>
(write <n>)))

Instantiation:
1: player A Jack
2: player A Janice
3: player B Sue
4: player B Jack
5: player B Sue

First outer iteration, <t> = B,
subinstantiation constrained to:
3: player B Sue
4: player B Jack
5: player B Sue

First inner iteration, <n> = Sue,
subinstantiation constrained to:
3: player B Sue
5: player B Sue

Second inner iteration, <n> = Jack,
subinstantiation constrained to:
4: player B Jack

Second outer iteration, <t> = A, etc.
```

Figure 3: A Set-Oriented Rule and its Iterations over PV Bindings

When there are nested foreach operators, the effect is compositional: Each iterator acts to reduce the size of the subinstantiation further by performing a selection as described above. In other words, when there are nested foreach operators, the innermost block is executed once for each combination of the distinct values of iterator variables, with the subinstantiation reduced to the result of
a selection specifying the iterator variables' attributes equal to their current values. This is illustrated in Figure 3, rule GroupByTeam, that only uses set-oriented PVs. This rule has a single instantiation. It uses nested foreach iterators to display all players on each team. However, since PVs are value-based, the two WMEs with value Sue are considered as part of the same subinstantiation. Hence, Sue will just be printed once for team B. Note how the current value of the team PV (<t>) constrains the domain of the name PV (<n>) in each iteration.

5.2. THE FOREACH OPERATOR ON SET-ORIENTED CES

The operation of foreach on set-oriented CES is similar to that described above for PVs but with two differences. First, the distinct values of the iterator variable now refer to the WMEs themselves. A convenient way to think of this is to imagine iterating over distinct time-tags and WMEs. During the execution of the foreach block, the iterator variable is a regular CE variable bound to a single WME. The second difference arises directly from this fact. Since, the iterator variable is bound to a specific WME, all of the set-oriented PVs that were referred to in that CE can only have a single value in their domain. Therefore, during the execution of the foreach block on a set-oriented CE, all the PVs referenced within that CE are treated as regular PVs.

Above, an analogy between foreach operators and the :scalar clause was established for set-oriented PVs. A similar one holds for the set-oriented CES. The subinstantiations formed by performing a foreach on a set-oriented CE are the same as the instantiations that would have been formed if that set-oriented CE had been designated a regular CE. However, those subinstantiations would have been separate instantiations.

6. Expressive Power of Set-Oriented Rules

The above description of the set-oriented rules has described their basic capabilities. However, it is only when seeing the succinctness with which frequently required operations can be expressed that their real utility is revealed. Set-oriented constructs enhance rule-based systems by allowing for the concise expression of processing unknown quantities of data, processing based on second order information, and hierarchical decomposition of data structures.

6.1 ITERATING OVER COLLECTIONS OF WMES

Unknown amounts of data stored in working memory are often processed through unbounded iteration in OPS5 programs [Cooper and Wogrin 1988]. When the iteration modifies the WMEs, state must be maintained to assure that the same WMEs are not modified repeatedly (e.g. by marking the WMEs as they are processed). Additional complexity is introduced by the requirement of several rules and the state they maintain. Set-oriented rules allow the entire collection of WMEs to be accessed within the execution of a single instantiation. The first set-oriented rule in Figure 4, SwitchTeams, updates collections of WMEs. The set-modify operation is used to express the conceptual unity of the operation of switching the members of the two teams. Without the set-oriented constructs, multiple instantiations are required and extra state must be maintained to record previously processed WMEs.

Counting WMEs can be accomplished by iteration. This cardinality can then be stored in another WME and used in subsequent LHS matching. However, the value is not automatically updated when the size of the collection changes. These difficulties are resolved by providing the ability to directly match second order information such as cardinality, as shown in Figure 4, SwitchTeams.

Below each rule, a specific instantiation is identified as the one that will be considered. Then for every iteration of the foreach operators, the instantiation is given constrained according to how the iterator will decompose it.
When there is a hierarchical information structure to be processed, several rules are needed and extra state must be maintained as the structure is traversed\(^3\). Set-oriented constructs allow of the WMEs to be matched in one instantiation and then hierarchically decomposed via the foreach iterator. GroupByA in Figure 4 prints out each member of Team A along with all of the members with whom they will have to compete. Certain hierarchical structures require transitive closure functionality in order to match the relevant WMEs. The specification of this functionality has not yet been investigated.

6.2 REMOVING DUPLICATE WORKING MEMORY ELEMENTS

The third rule in Figure 4, RemoveDups can only be accomplished in regular OPS with great difficulty: reducing working memory to a set\(^4\). This rule finds instances of multiple team players with the same name and same team and then deletes all but the most recent one. Notice that there will be one instantiation of this rule for each player-team pair occurring in multiple WMEs. An alternative formulation, AlternateRemoveDups, simply matches all player-team pairs and iterates over the values to remove redundant elements. However, this rule cannot discern whether any duplicates exist, thus its instantiation can fire unnecessarily.

7. Summary

Set-oriented constructs bring rule-based languages closer to database systems by moving them from tuple-based processing to set-based processing, allowing many commonly performed tasks to be concisely and efficiently specified. This integration shows promise in adding both expressivity and efficiency to expert system and database programming languages.

References


\(^3\) SOAR uses multiple WMEs to describe a single data structure [Laird et al. 1986]. Using OPS5 rules, it is not possible to access a complete SOAR data structure since the number of WMEs it contains is not fixed.

\(^4\) In order to fully appreciate the set-oriented solution, the reader is encouraged to attempt to express this task in regular OPS5.